GEL: Grid Execution Language

CHUA Ching Lian, Francis TANG[*], Praveen ISSAC, Arun KRISHNAN[†]
Bioinformatics Institute, Matrix #07-01, 30 Biopolis St., Singapore

March 16, 2005

Abstract

We consider the problem of programming parallel applications for a Grid environment, in the presence of the two main challenges (i) high-latency communications and (ii) heterogeneity.

We describe a new scripting language, GEL, whose semantics have been designed for execution on a heterogeneous, distributed computer. The language provides syntactic constructs for while loops, conditionals and explicitly parallel execution. The language is designed to work well given these two challenges, and to allow succinct representation of parallel programs, resulting in easier-to-maintain code.

The programs can use legacy applications without re-engineering, and do not explicitly refer to resource names or use middleware-specific references. This middleware-independence allows us to execute the same script on an SMP machine, cluster or Grid.

We describe three example applications written in GEL: an optimisation problem solved using a swarm algorithm; an allergenicity prediction pipeline; and transcript analysis for tissue-specific gene expression. We have run these scripts unchanged on an SMP machine, on PBS, SGE and LSF clusters, and on a Globus-based Grid.

1 Introduction

As the list of large-scale grid projects grows rapidly (see, for example, Grid Physics Network (GriPhyN) [16], EU DataGrid [1], and NASA Information Power Grid (IPG) [20]), increasingly many users have vast resources at their disposal. However, tools allowing users to effectively utilise such resources are scarce. Developing Grid applications using the core Globus Toolkit, the de facto standard Grid middleware, is far from straightforward.

Consider the issues of geographic separation and heterogeneity of Grid compute resources. The former rules out any program, which relies on fine-grained parallelism, as a viable Grid application. The latter, however, if exploited correctly, can provide savings in compute, storage and software resources, rather than being a hindrance. Executing non-communicating programs concurrently across different Grid nodes, and exchanging input/output files between grid nodes before and after execution of component programs, gives a coarse-grained parallel application which can exploit both characteristics.

Scripting languages, such as bash or tcsh, are commonly used to pull together several programs into one application. However, since these scripting languages were not designed for Grid programming, the programmer must explicitly call commands to transfer files and submit programs as jobs for execution. Furthermore, the programmer must implement his own code to impose control flow of programs, e.g. ensure barriers are respected, and schedule jobs (which includes compute resource selection). (See Fig. 6 for an example of such a script.)

Commonly used patterns from these scripts can be factored out and implemented either as extensions (e.g. subroutines) or in a new programming/scripting language. In the latter approach, dependencies between jobs can be implicit in the syntactic structure

[*]Chua C. L. and F. Tang contributed equally to this article.
[†]Correspondence should be addressed to A. Krishnan.

of the script, without the need for explicit naming and referencing of dependent jobs. It can also allow for better scheduling, since, at the time of execution, scheduling decisions can be made from a global perspective, i.e. by analysis of the whole script.

Examples of the latter approach include APST [8], DAGMan [26], GridAnt [7] and Pegasus [12][1]. APST, DAGMan and GridAnt provide language support to initiate file transfers and job submissions, and provide basic control-flow constructs, such as sequential dependencies between jobs, so long as the dependencies are *acyclic*. In particular, there is no support for iterative loops.[2] Their scripts are *concrete* in the sense that they require explicit naming of resources, i.e. host and service names. In comparison, Pegasus can take *abstract* scripts, i.e. scripts without explicit mention of resource/service names, and map them to concrete DAGMan scripts. Pegasus scripts are also limited to only acyclic dependencies between jobs.

Our scripting language, GEL, improves on the languages mentioned above by allowing the programmer to express *abstract* scripts with *cyclic* dependencies, such as while loops. Furthermore, dependencies between jobs are implicit in the syntactic structure of the script which results in scripts that are easier to maintain. We have successfully implemented a parameter estimation application using an iterative, evolutionary algorithm based on particle swarm optimisation. Furthermore, since the scripts are abstract, they can be executed using different interpretors for different platforms, without source-level changes. Our current interpretor implementation is factored into DAG *builder* and DAG *executor* components. We currently have five *executors*: one each for SMP machines[3], Sun GridEngine (SGE), LSF and PBS clusters, and Globus Grids (see Sec.3.3).

The rest of the article is organised as follows. In Sec. 2 we discuss the challenges posed by Grid computing, with particular emphasis on programming of Grid applications. Section 3 describes the concepts of GEL and some of its current interpretor implementations. We illustrate with three concrete examples: Sec. 4 describes an optimisation problem solved using an evolutionary algorithm (i.e. using iteration); Sec. 5 describes a pipeline for prediction of allergenic proteins; and Sec. 6 describes a pipeline for analysing tissue-specific transcripts. We conclude and describe future work in Sec. 7.

[1] APST, DAGMan, GridAnt and Pegasus explicitly specify dependencies between jobs.

[2] Only early implementations of GridAnt which were based on the Ant execution engine have this limitation.

[3] In particular, a single-processor machine is considered as a 1-way SMP.

2 Some Challenges in Grid Programming

The prevailing parallel-programming formalisms, such as MPI, PVM and OpenMP, are best suited to tightly-integrated parallel computers (e.g. shared-memory computers). Grids, however, are typically more loosely connected—sometimes separated by geographically large distances—resulting in poor inter-node communications performance, especially in terms of latency.

As a distributed computer, a distinguishing characteristic of a computational Grid is that it is heterogeneous. One view of heterogeneity is that it is a necessary evil to allow for the integration of existing computational hardware/software and future extensibility of the resulting set up. However, we believe heterogeneity should be embraced since it allows subtasks to be run on different nodes, depending on suitability. For example, consider the following areas of resource heterogeneity.

- Hardware capacity: some Grid nodes have more memory or scratch space than others.
- Hardware architecture: some subtasks might run more efficiently on a vector-processing machine, whereas others might run more efficiently on scalar-processing machines.
- Data: it might not be feasible to replicate large databases (e.g. genomic databases) across all Grid nodes, in which case subtasks should run on Grid nodes which are "close" to the data on which they depend.

- Software: for software that require licencing, it would be prohibitively expensive to purchase licences for all compute resources.

Legacy applications, i.e. applications that are in use (and work well), are present in various computing fields. In particular, bioinformaticians have amassed large collections (e.g. EMBOSS [34]) of software tools and utilities which are used on a daily basis. Users (and thus programmers) are often hesitant to port or re-implement their programs to make them Grid-enabled. Furthermore, given the sheer volume of existing applications, porting is a tremendous effort in itself. This is the main emphasis of the GriddLeS project [3].

As developers race to get their programs Grid-enabled, they often find that the resulting code becomes middleware-specific, e.g. Globus [17], LSF, SGE, or PBS. Not only does this present problems when moving between middleware implementations, it also slows down the development process since the underlying middleware must be set-up and configured, simply to test and develop code. For example, short of providing each team member with his/her own testbed, it would be extremely hard to test development code in isolation.

The next section describes the design decisions in GEL made to address specifically these problems of high latency communications, heterogeneity, legacy applications and portability. We assume that security of computations spread over wide area networks is handled by the underlying middleware, and though fault-tolerant recovery is an important issue in Grid programming, in this article, we consider only basic recovery.

3 Grid Execution Language

The high-latency of communications broadly rules out fine-grained parallelism as a viable way to use the Grid. Instead, the Grid is best utilised by coarse-grained parallel programs, where subtasks are run on different machines, in parallel where possible. Services-oriented workflows are examples of such coarse-grained parallel programs. However, workflows can also be implemented using a scripting language similar to bash, perl and python. The design of such a Grid scripting language should have the following salient features, keeping in mind the Grid-programming challenges delineated above.

- Its semantics should be designed, from the outset, with execution on a heterogeneous, distributed computer in mind.
- Its programs should have enough information to allow for its subtasks to be scheduled in a way that effectively uses the Grid resources.
- The syntax should be middleware independent, and as such the programs should be able to run on different middleware implementations[4], provided a suitable interpretor is available.

The last point is straightforward. We choose to implement the language as an interpreted language (cf. bash, perl or python) and programs written in our language should have no middleware-specific references. Instead, we push all the middleware dependencies into specific interpretor instances. For example, we have one interpretor instance which uses GridFTP commands to stage files and Globus GRAM [11] requests to execute jobs, and another instance which assumes a campus-wide NAS-based file system and uses SGE to queue jobs for execution (see Sec.3.3).

We address the other two points in more detail below.

3.1 Language Syntax and Semantics

Semantically, executing a GEL script amounts to executing a collection of binaries in the specified order. Conceptually, all binaries run in the same working directory where they read, create and modify files. The output of the program consists of the remaining files at the end. In practice, some binaries can be run in parallel and in separate directories on different computers; the interpretor must provide the illusion that

[4]In this article, we use the term *middleware implementation* to refer generically to the underlying software platform, rather than specifically *Grid* middleware. In particular, this includes the OS and scheduler for clusters, and the bare OS itself for SMP machines.

they run in the same working directory, by copying files between working directories.

The basic atomic components of our programs are Jobs. Informally, a Job specifies a binary along with other information such as its requirements (e.g. processor/OS architecture and memory) which allows the interpretor to correctly stage the required files and execute the binary (be it through an "exec" OS call, SGE qsub, Globus GRAM request, or otherwise). Executing a Job amounts to possibly staging more input files into the working directory, and then running the binary. The files present in the working directory after the binary terminates are deemed to be the output of the Job.

Inspired by the design of Globus RSL scripts, we declare Jobs by defining a combination of *atomic predicates* (e.g. of the form $(exec = \texttt{bl2seq})$, $(arch = \texttt{ia64})$ and $(mem \geq \texttt{100M})$) using conjunctive ($\wedge$) and disjunctive ($\vee$) operators, giving a *predicate*. Other examples of predicates might impose constraints on required software licences/libraries, required data, cpu count and required scratch-space.

For example,

$$\begin{array}{ll} & (exec = \texttt{bl2seq}) \\ \wedge & (dir = \texttt{ia64}) \\ \wedge & (arch = \texttt{ia64}) \\ \wedge & (args = \texttt{-i seq1.fasta -j seq2.fasta}) \end{array} \tag{1}$$

specifies a Job whose binary `bl2seq` is located in directory `ia64` and requires an ia64 architecture machine to run. Supposing we also have a binary `bl2seq` in directory `sun`, which requires a `sparc9` architecture, and which is functionally equivalent to the previous binary, then the Job can be specified as

$$\begin{array}{l} \left(\begin{array}{ll} & (dir = \texttt{ia64}) \wedge (arch = \texttt{ia64}) \\ \vee & (dir = \texttt{sun}) \wedge (arch = \texttt{sparc9}) \end{array} \right) \\ \wedge\, (exec = \texttt{bl2seq}) \\ \wedge\, (args = \texttt{-i seq1.fasta -j seq2.fasta}) \end{array}$$

which informally states that to execute this Job, you can execute either binary with the same arguments.

For flexibility, we allow parameterisation of arguments by way of a function/procedure mechanism. For example, we can name a parameterised Job as

$$\begin{array}{lll} B(a,b) := & & (exec = \texttt{bl2seq}) \\ & \wedge & (dir = \texttt{ia64}) \\ & \wedge & (arch = \texttt{ia64}) \\ & \wedge & (args = \texttt{-i}\ a\ \texttt{-j}\ b) \end{array}$$

in which case $B(\texttt{seq1.fasta}, \texttt{seq2.fasta})$ "instantiates" to Job (1).

Programs consist of combinations of Jobs using a collection of language constructs, some sequential and some parallel. The sequential constructs are standard and are a common feature of many programming languages: sequential composition $P.Q$ (execute P, then execute Q); loop iteration $\mathsf{while}\ A\ \mathsf{do}\ P$ (continually execute P depending on the outcome of executing A); and conditional $\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q$ (execute A and, depending on its outcome, execute either P or Q). Conceptually, executing a program $P.Q$ involves executing P and then executing Q, in the same directory; if P and Q run on different hosts, then we must copy files from the working directory of P to that of Q.

The parallel constructs are perhaps less familiar, and we describe them in more detail. The most basic parallel construct is the pair-wise parallel composition operator $-+-$ (cf. the pair-wise sequential composition operator $-.-$).[5] Given two programs P and Q, their parallel composition is written $P+Q$. By writing $P+Q$, the programmer is merely providing a hint to the interpretor that P and Q are independent of each other; P and Q do not interfere with each other by creating/modifying files of the same name, nor does the behaviour of one depend on files created/modified by the other, and vice-versa. Thus, the interpretor can (*possibility*), but is not obliged (*non-obligation*) to, execute P and Q,

[5]The choice of $-.-$ for sequential composition and $-+-$ for parallel composition are suggestive of the two ring operations: both are associative (i.e. $(P.Q).R = P.(Q.R)$ and $(P+Q)+R = P+(Q+R)$) and parallel composition is commutative (i.e. $P+Q = Q+P$). Furthermore, for deterministic programs, sequential composition is left distributive over parallel composition (i.e. $P.(Q+R) = P.Q+P.R$). However, it is not right-distributive, (i.e. $(P+Q).R \neq P.R+Q.R$). Associativity of the two operators means that there is no ambiguity in expressions of the form $P.Q.R$ and $P+Q+R$, thus the notations $\prod_i P(i)$ and $\sum_i P(i)$ are well-defined.

(i) concurrently, and (ii) in the same working directory. In point (i), *possibility* allows us to take advantage of multiple processors; *non-obligation* allows for single threaded implementations of an interpretor. In point (ii), *possibility* allows for optimisations on machines with a shared disk image, e.g. SMP and cluster machines; *non-obligation* allows Jobs to run on different Grid nodes. Executing $(P+Q).R$, means that R can only be run after the last of P and Q has finished, i.e. there is an implicit barrier.

The parallel composition of programs is the basic mechanism allowing us to express parameter-sweep-style programs in our language. For example, suppose $A(-)$ is a parameterised Job that takes one parameter, and we want to run the program for arguments 1..20. We use the parallel-for construct

$$\sum_{x=1}^{20} A(x)$$

which is behaviourally equivalent to

$$A(1) + A(2) + \cdots + A(20) \ .$$

Another useful parallel construct is

$$\sum_{f \in \{\texttt{gene*.fasta}\}} B(\texttt{q.fasta}, f)$$

which, when executed, finds all files in the working directory matching *glob* pattern `gene*.fasta` and instantiates one copy of $B(-,-)$ for each file. For example, if the matching files are `gene1.fasta`, `gene2.fasta`, `gene3.fasta` then this is behaviourally equivalent to

$$\begin{array}{rl} & B(\texttt{q.fasta}, \texttt{gene1.fasta}) \\ + & B(\texttt{q.fasta}, \texttt{gene2.fasta}) \\ + & B(\texttt{q.fasta}, \texttt{gene3.fasta}) \end{array}$$

The semantics are designed in such a way that it is straightforward to execute programs over Grids, and also allows for optimisations on SMP and cluster machines.

3.2 Interpretor architecture

In the interests of portability, our programs should not contain network-specific references (e.g. host-names). Thus, to effectively use Grid resources, we must schedule/allocate resources to Jobs.

Other than the explicit annotations associated with Jobs (e.g. required architecture, memory, scratch space, libraries, software and data) there is also implicit dependency information in the language constructs. This information could help the interpretor make better scheduling decisions.

For example, suppose A is a preprocessing step to generate the input files for $B(1), \ldots, B(N)$, for some fixed N, and these Jobs are composed in a pipeline as follows:

$$A.\sum_{i=1}^{N} B(i) \ ,$$

that is, execute A, and then in parallel execute $B(1), \ldots, B(N)$. Since $\sum_{i=1}^{N} B(i)$ cannot execute before A completes, if we were to schedule A without considering the instances of $B(-)$, we would likely run A on the machine which has the least utilisation. However, if we also consider the instances of $B(-)$, we might try to run A on a large Beowulf cluster so that we won't need to perform inter-Grid-node file staging before executing each instance of $B(-)$.

If N is so large (e.g. 10,000 or more) and we would like to run the instances of $B(-)$ on more than one cluster, we could run $\sum_{i=1}^{M} B(i)$ on one node, and $\sum_{i=M+1}^{N} B(i)$ on another node, for some M between 1 and N. Or we could even transform[6] the program into

$$\sum_{i=1}^{N} A.B(i)$$

if we are confident that A is deterministic. This may give us better performance in the case where we partition the instances of $A.B(-)$ across many Grid nodes. Of course, such scheduling is only possible if the interpretor has sufficient information.

To preserve *all* information, the interpretor should take, as input, the whole program. However, we believe this introduces its own (unnecessary) complications: the interpretor would have to not only support

[6]Note the similarity of the transformation from $A.\sum_{i=1}^{N} B(i)$ to $\sum_{i=1}^{N} A.B(i)$, and the *code motion* optimisations [5] of traditional compilers.

all the syntactic nuances of the programming language, but also be able to deal with apparent cyclic dependencies (e.g. the dependencies present in the while loop). Furthermore, since interpretors must use specific knowledge of the architecture of the host computer, we expect many different implementations, and each implementation would have to be able to parse, analyse and interpret programs. So, in the interests of (code) maintainability, we propose a compromise which separates the scheduling responsibilities from the interpretor itself. Instead, we introduce an intermediate description of programs which is essentially that of Job instances and their (acyclic) dependencies on other Job instances, i.e. DAGs. These DAGs give more information than simply atomic Jobs by themselves, yet at the same time eases from the scheduler, the burden of the syntactic analyses required for whole programs.

The apparent cyclic dependencies in constructs such as the while loop are avoided by progressively (i.e. at runtime) unwinding the while loop and passing the DAGs to the scheduler in a piece-wise fashion. More explicitly, to execute

$$(\mathsf{while}\ A\ \mathsf{do}\ P).Q\ , \tag{2}$$

we first submit an instance of A, which we call A_1 to the scheduler, and then wait for its completion. When A_1 finishes, if it evaluates to true, then we submit instances of P and A, which we name P_2 and A_3, where P_2 depends on A_1 and A_3 on P_2 (that is, A_3 should not execute until P_2 has completed), and we repeat this for A_3. When some A_j eventually evaluates to false, we simply continue by submitting instance Q_{j+1} to the scheduler. The resulting DAG that is submitted is of the form

$$A_1,\ P_2(A_1),\ A_3(P_2),\ \ldots,\ A_i(P_{i-1}),\\ P_{i+1}(A_i),\ \ldots,\ A_j(P_{j-1}),\ Q_{j+1}\ ,$$

where the instance name in parentheses denotes dependent Jobs, and Job instances are given unique subscripts. To give another example, executing

$$\left(\sum_{i=1}^{5} P(i)\right).B$$

gives the following DAG

$$P_1,\ \ldots,\ P_5,\ B_6(P_1 \cdots P_5)\ .$$

What appeared to be a cyclic dependence between A and P in Ex. 2, disappears when we consider dependencies between Job instances. The interpretor submits as many Job instances with dependencies as possible to the scheduler. The interpretor will get "stuck" at constructs such as $\mathsf{while}\ A\ \mathsf{do}\ P$, $\mathsf{if}\ A\ \mathsf{then}\ P\ \mathsf{else}\ Q$ and $A.\sum_{f\in\{F\}} P(f)$, since in each of these cases, execution depends on the result of executing A.

Thus we factorise the monolithic scheduler which runs programs directly, into (1) a (DAG) *builder* which translates from programs to the intermediate DAG form, and (2) a (DAG) *executor* which runs DAGs on some target class of host.

3.3 Implementation Status

Currently, there are implementations of interpretors which allow GEL scripts to be run on an SMP machine, on a cluster running PBS, LSF or SGE, and on a Globus-based Grids. Standard binaries and MPI Jobs are supported.

The SMP executor runs Jobs on the same computer running the interpretor using the OS exec call. There is a user-specified limit on concurrently running Jobs. The user can use this limit to constrain the amount of resources used by specifying fewer concurrent Jobs than there are cpus. Alternatively, the user can improve performance in cases where the Jobs cannot saturate the cpu by specifying more concurrent Jobs than cpus.

The cluster implementations run Jobs by calling a *job submit* command (e.g. `qsub` and `bsub`). These implementations assume that the configuration is homogeneous: same cpu/OS platform and same software configuration for all compute nodes, and also a shared file system.

The Globus Grid implementation runs Jobs on Grid nodes using the Globus GRAM protocol. It assumes that there is no shared file system and so it must transfer files using the GridFTP protocol. It allows for software heterogeneity, but there is currently no support for hardware/platform heterogeneity: it is assumed all execution hosts are of the same ar-

chitecture and OS, e.g. i386-linux. The disjunctive operator for Job definitions is not available since it has little use in an almost homogeneous environment. Condor-based Grid support is currently being implemented.

All interpretors are included in the downloadable distribution. Installation does not require super-user privileges; the user can install in his home directory, and the interpetors only require a modern Java virtual machine to run.

The current Globus Grid implementation uses a basic round-robin scheduler. We can implement more advanced scheduling, for example similar to that used in the GrADS project [10]. Alternatively, there is the possibility to run Jobs on the Grid indirectly via a metascheduler such as Condor-G [26], Nimrod [4] or APST [8].

A GUI front-end called *Wildfire* is also available. This allows the user to construct workflows using a drawing-based metaphor and is already preconfigured with Bioinformatics templates for creating Job definitions. *Wildfire* exports GEL scripts and invokes the relevant interpretor depending on the execution host.

4 Particle Swarm Optimisation Example

We now illustrate the ideas presented above by describing a concrete example. For many non-linear optimisation problems, traditional techniques cannot find optimal solutions for various reasons, be it because the problems have no closed-form analytical equations, or because they are ill-conditioned. Particle Swarm Optimisation [14] is an optimisation heuristic based on simulation of social behaviour, observed in, for example, bird flocking.

Swarm algorithms use the concept of leaders-followers information exchange. On each evolution, two groups, leaders and followers, are selected. Individuals are ranked according to their fitness. A set of highest ranking individuals are then chosen to be the leaders and the rest become followers. Leaders are those individuals who have a higher than average measure of fitness. They do not move from their location at the current evolution so as to ensure elitism. As for the followers, each one acquires information about its neighbouring leaders, and identifies for itself the leader to follow. Next it updates its *flying* direction according to both the selected leader's *flying* direction and its previous direction. Finally, it moves to a new location with the new information. This process is repeated again until an optimal solution is found.

In the swarm algorithm each individual performs computations independently after obtaining information about the leaders. Hence it can benefit from parallel execution.

4.1 Workflow

Our example is an implementation of a swarm algorithm by Ray et al. [33]. We applied the algorithm to a parameter estimation problem for a model of biochemical pathways comprising 36 unknowns and eight ODEs [32].

The model is formulated as a nonlinear programming problem with differential-algebraic constraints. Due to the very nature of such types of problem, most of them are multimodal and global optimisation techniques are more suited to finding an optimal set of solutions. As such swarm intelligence is used here to find a global optimal solution.

The algorithm is presented diagrammatically in Fig. 1. We divided the algorithm into separate components (`init`, `eval1`, etc.) based on their functions, and composed the modules into a script, displayed in Fig. 2. The equivalent bash script is displayed in Fig. 6. Note that in bash, we must: (1) explicitly name the Job instances; (2) explicitly handle the sequential dependencies between modules by passing dependent Job information to the SGE qsub command or by implementing lock files.

Furthermore, the bash script in Fig. 6 is specific to SGE because of the qsub command and its command-line arguments. Even if we parameterise the qsub command, the script still assumes that the compute hosts mount a shared file system, and thus still more work is required to port it to a Globus Grid.

In contrast, we can execute the GEL script in Fig. 2, without any changes, using any interpre-

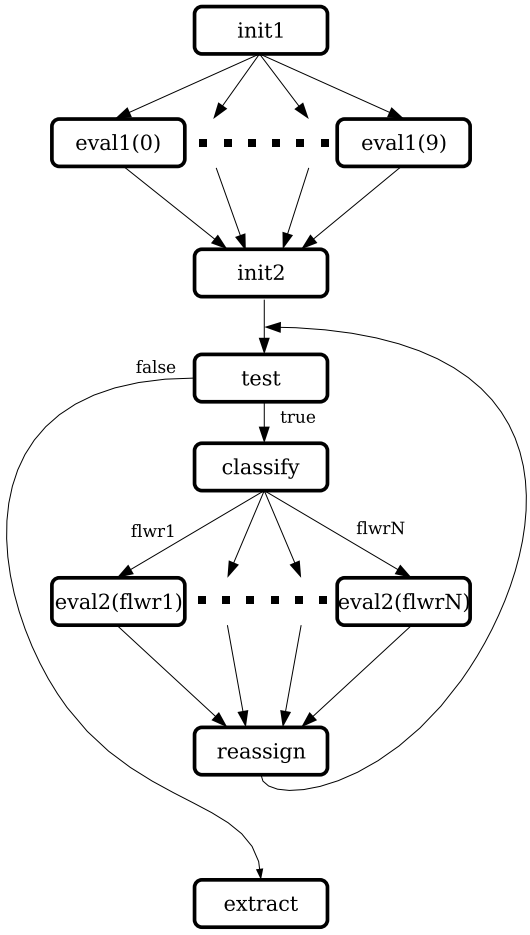

Figure 1: Diagram of swarm optimisation example. Job `init1` generates an initial population; `eval1` evaluates each individual in parallel; and `init2` collates the results of the evaluations. Job `test` encapsulates the termination condition for the optimisation. Job `classify` selects the leaders and followers for the next evolution; `eval2` evaluates the individuals; and `reassign` collates the results for the next evolution. Finally, `extract` collates and formats the final result.

```
 1: init1:=     {exec= "prog1";
 2:              dir= "swarm";
 3:              ipdir = "init" }
 4: eval1(i):=  {exec= "prog2";
 5:              args= $i;
 6:              dir= "swarm"}
 7: init2:=     {exec= "prog3";
 8:              dir= "swarm"}
 9: test:=      {exec= "prog7";
10:              dir= "swarm"}
11: classify:=  {exec= "prog4";
12:              dir="swarm" }
13: eval2(i):=  {exec= "prog5";
14:              args= $i;
15:              dir= "swarm"}
16: reassign:=  {exec= "prog6";
17:              dir= "swarm"}
18: extract:=   {exec= "prog8";
19:              dir= "swarm";
20:              cmdir= "result"}
21:
22: init1 ;
23: pfor i = 0 to 9 do
24:     eval1($i)
25: endpfor ;
26: init2 ;
27: while test do
28:     classify ;
29:     pforeach file of "follower_sol*" do
30:          eval2($file)
31:     endpforeach ;
32:     reassign
33: endwhile ;
34: extract
```

Figure 2: Swarm in GEL. A typical GEL script consists of definitions of Jobs, followed by the control-flow description. Here we see the concrete syntax for two types of parallel for (`pfor` and `pforeach`), the while loop, parameterised Job invocation and sequential composition ("`;`").

tors implementation, from SMP machine, to homogeneous cluster and to a Globus Grid.

4.2 Script

Figure 2 shows the basic layout of a GEL script, namely a list of Job definitions, followed by the control-flow description.

The syntax for Job definition predicate uses a semicolon-delimited list of atomic predicates, e.g. `exec="prog1"` and `dir="swarm"`, within a pair

Attribute	Description
`exec`	File name of executable
`exectype`	Job type: `mpi` or normal (blank)
`nproc`	Number of cpus required (for MPI Jobs)
`dir`	Local directory in which executable resides
`args`	Command line arguments to be passed to executable
`ipdir`	Local directory containing read-only files
`cmdir`	Local directory containing read-write files
`software_req`	Software packages required to run this Job

Table 1: Attributes for atomic predicates in Job definitions.

of braces to represent a conjunction[7]. The attributes used in our example are described in Tab. 1. The *common directory* (`cmdir`) serves two purposes: it tells the interpretor (1) in which directory to find input files, and (2) to which directory the output files should be staged after execution of the Job. In this example, we use a common directory to specify the local directory in which to store the contents of the working directory after executing `extract`.

Other supported attributes include `ipdir` and `software_req` which allow the programmer to specify an *input directory* and *software requirements* (see Sec.5). The input directory is a local directory whose contents are staged to the execution host before execution. The executables should not modify files populated from this input directory. This distinction allows for future optimisations in case large, read-only input files are required for Jobs.

Tightly-coupled parallel Jobs implemented as MPI programs are supported via the attributes `exectype` and `nproc`. The programmer can use the first to specify that the executable is an MPI program (and thus should be run in the correct way) and the second to specify how many cpus to be used.

Job `eval2` is a parameterised Job with one parameter `i`. We follow standard convention and interpret this to mean that `i` is assigned whatever value is provided at the time of invocation (we later invoke it as `eval2($file)`, i.e. variable `i` will take whatever value is assigned to variable `file`). We prefix variable names with a dollar symbol ($) to denote its value. Thus in the case of `eval2`, the command-line argument provided to `prog5` is the value of `i`.

The second part of the script describes the flow of execution of the program. Jobs are invoked by name (e.g. `init1`), possibly with arguments delimited by parentheses (e.g. `eval2($file)`). Though not shown here, the syntax also supports multiple arguments. The example shows the concrete syntax of the while loop, and parallel for operators:

$$\sum_{i=a}^{b} \mathtt{P}(i) \quad \text{and} \quad \sum_{f \in \{F\}} \mathtt{P}(f)$$

are written as

`pfor i =` a `to` b `do P($i) endpfor`

and

`pforeach f of` F `do P($f) endpforeach`

respectively.

Executing the program entails the following. We first copy the contents of `init` into the working directory. We then execute `init1`, before executing, in parallel, a copy of `eval1` for `i` from 0 to 9. After the parallel copies of `eval1` have completed, we execute `init2`. Next, provided `test` returns true, we execute the body of the while loop, that is, we execute `classify` and then, again in parallel, a copy of `eval2` for each file matching `"follower_sol*"` in the working directory. We continue to execute the body of the while loop provided the test returns true. When `test` finally returns false, we execute `extract`, before copying the contents of the working directory into `result`.

Returning to Fig. 2, the first `pfor` loop initialises the swarm individuals starting from some random values. The `while` loop is used to iterate over the evolutions, i.e. its body specifies the required stages

[7] Disjunction will be added in the future.

for each evolution. The `pforeach` loop executes the independent code for each follower, evaluating and updating the flying direction. This loop construct is used because on each evolution, the set of leaders and followers changes. The `pforeach` captures these changes by matching a different number of files on each evolution, and executing an instance of the statement in the body for each file.

The `pfor` loop behaves similarly to the `pforeach` but runs parallel copies of the statement it its body by stepping from the starting integer to the ending integer in steps of one. For this program, we have used a swarm with a population of 10. A further difference between the two parallel loop constructs is the variability in `pforeach`; in a `pfor`, the number of parallel copies is known to the interpretor before execution starts, but in the other, the number of parallel copies can only be known after the Job preceding the `pforeach` has completed.

The `while` loop examines the standard output of `test` to decide on whether to step into the loop body. The boolean test is false if `test` writes to standard output and true if nothing is written. This same boolean testing mechanism is used in the conditional construct (if). In this example, `test` tests for convergence of the solution, allowing the loop to terminate.

We also have concrete syntax for other constructs. The general parallel composition operator $(- + -)$ is written as a pipe (`|`)[8]. When we write `P|Q`, we simply mean that there are no dependencies between subprograms `P` and `Q`, and so they can be executed in parallel. The if-then-else construct is written as `if A then P else Q endif`.

The constructs can be combined together to obtain programs such as:

```
A | (B;C) | D
```

[8]The pipe and similar-looking symbols are commonly used to represent parallel execution in the process algebra community [28, 29, 30, 18]. However, it should not be confused with the pipe syntax as used in unix shells.

Interpretor	cpus	time
local	1	357min
Globus Grid	16	89min
SGE	8	123min
bash+SGE	8	122min

Table 2: Performance of various interpretors. During runtime, the idle cpus can be used by other users; the interpretors do not monopolise all the cpus displayed in the table above.

Name	Nodes	cpus	Scheduler
turing	4	8	SGE
church	2	4	PBS
goedel	2	4	LSF

Table 3: Grid testbed configuration. The testbed consists of three Pentium III 800MHz clusters running Linux and Globus Toolkit 2.2.4.

and

```
if A then
  (while B do (C;D) endwhile)
else
  (E|F)
endif
```

4.3 Performance results

The script in Fig. 2 was run using our three different interpretors, the results of which are tabulated in Tab. 2. The local execution interpretor was the slowest, as expected. The Grid version, run across all three clusters totalling 16 cpus (see Tab. 3) was the fastest, even with naive round-robin scheduling. The SGE version, run on turing (see Tab. 3) was slower than the Grid version. We compared our SGE-targeted interpretor against the hand-crafted bash script displayed in Fig. 6 and found the performance to be almost identical. Thus we can conclude that any overheads introduced by our scripting language/interpretor have a negligible impact on performance.

We also note that during the critical, non-parallel parts of the script, the unused cpus can be used by

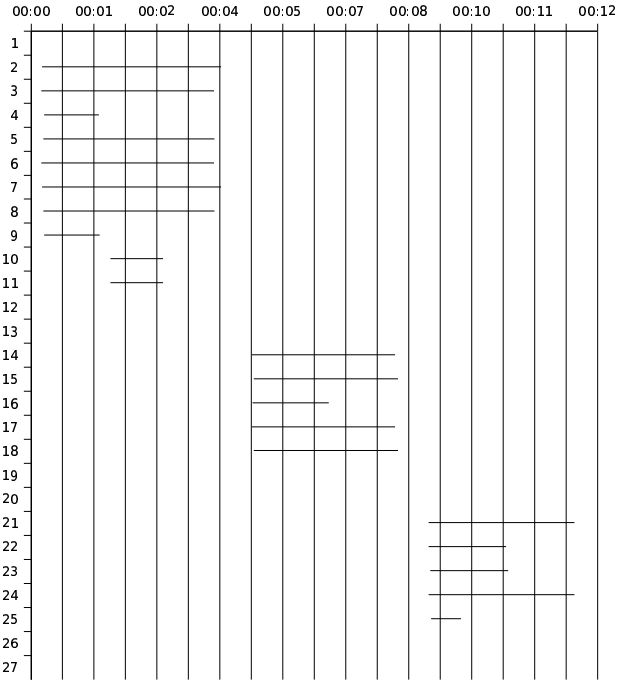

Figure 3: Profile of Job execution. Time (hh:mm) is shown along horizontal scale. The execution of each Job is shown as a line. Total cpu time is 54m 40s and wall time is 12m 55s.

Jobs from other users. That is, the interpretor does not monopolise the resources from the beginning, unlike, for example, typical MPI programs. Hence, the parallel speed-up ratio (i.e. speed-up divided by number of processors), is not indicative of efficiency. Furthermore, since the user does not need to request a fixed number of cpus before running the script, execution starts as soon as the first Job can run, and the parallel regions of the script can use more cpus as they become available. Figure 3 shows a profile of the execution of each Job for a small example run on turing (8 cpu cluster). The consequences of barriers are clearly shown (e.g. Jobs 14–18, and 21–25).

5 Allergenicity Prediction Example

Allergens are proteins that induce allergic responses. More specifically, they elicit IgE antibodies and cause the symptoms of allergy, which has been a major health problem in developed countries. With many transgenic proteins introduced into the food chain, the need to predict their potential allergenicity has become a crucial issue. Bioinformatics, more specifically, sequence analysis methods have an important role in the identification of allergenicity. One approach to allergenicity prediction is to determine, automatically, motifs from sequences in such a database, and then search for these motifs in the query sequences.

5.1 Workflow

The workflow [24], shown in Fig. 4, can be summarised as follows. A database of allergens is clustered based on their pairwise distance. We then use wavelet analysis [21] to identify the motifs in each cluster. We then look for these motifs in the query sequence.

We use *ClustalW* to generate the pairwise global alignment distances between the known allergenic protein sequences. We then use these distances to cluster the sequences by partitioning around medoids using the statistics tool *R Project* [2]. Each cluster of protein sequences is subsequently aligned us-

ing *ClustalW*, and we use wavelet analysis on each aligned cluster to identify motifs in the protein sequences. HMM profiles [13, 15] are then generated for each identified motif. We use these profiles to search for the motifs in each query sequence, and thus predict whether it is an allergen or not.

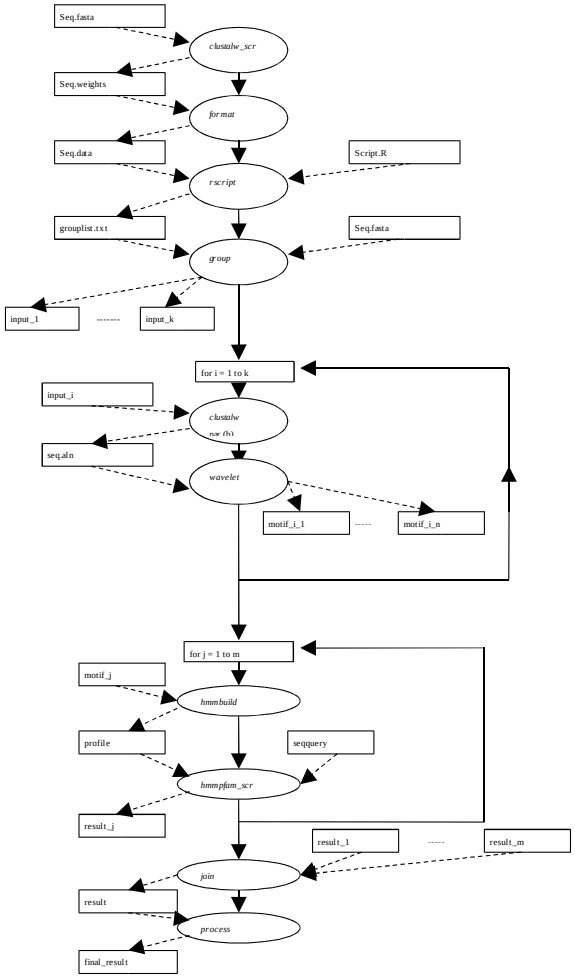

Figure 4: A pipeline for allergenicity prediction. The ovals and solid arrows correspond to execution units and execution flow. The rectangles and dashed arrows correspond to data files and data flow.

```
 1: A:={  exec= "clustalw_scr";
 2:       dir= "clustalw/";
 3:       ipdir= "clustalw/"}
 4: B:={  exec= "format";
 5:       args= "Seq.weights","Seq.data";
 6:       dir= "format/"}
 7: C:={  exec= "rscript";
 8:       software_req= "R";
 9:       dir= "R/";
10:       ipdir= "R/Rfiles/"}
11: D:={  exec= "group";
12:       args= "Seq.fasta","grouplist.txt";
13:       dir= "group/"}
14: E(i):={exec= "clustalwinput.sh";
15:       args= =$i;
16:       dir="clustalw/"}
17: F:={  exec= "wavelet_scr.sh";
18:       args= "seq.aln";
19:       dir= "wavelet/";
20:       ipdir= "wavelet_files/"}
21: G(j):={exec= "hmmbuild";
22:       args= "profile",$j;
23:       dir= "hmmbuild/"}
24: H:={  exec= "hmmpfam_scr";
25:       args= "profile","result.txt";
26:       dir= "hmmbuild/";
27:       ipdir= "hmmpfam_files/"}
28: I:={  exec= "join_scr";
29:       args= "result";
30:       dir= "join/"}
31: J:={  exec= "process_scr";
32:       args= "result","final_result";
33:       dir= "process/";
34:       ipdir= "process/binary/";
35:       cmdir= "result/"}
36:
37: A; B; C; D;
38: (pforeach file of "input*.txt" do
39:     E($file); F
40: endpforeach);
41: (pforeach file of "*motif*" do
42:     G($file); H
43: endpforeach);
44: I; J
```

Figure 5: Script for allergenicity prediction. This is the script corresponding to the pipeline displayed in Fig. 4.

5.2 Script

Informally, the interpretor executes the example script in Fig. 5 as follows: first execute `A`, `B`, `C` and `D` in order, then execute, in parallel for each file matching `input*.txt`, a copy of `E` followed by `F`; then execute, again in parallel, a copy of `G` followed by `H` for each file matching `*motif*`; finally execute `I` followed by `J`. The execution is faithful to the flowchart in Fig. 4.

As seen in the previous example, we use the *parallel for* construct in lines 38 to 40, and again in lines 41 to 43. Recall that this hints to the interpretor that the body of the loop is independent for each instantiation of the loop variable (`$file` in both cases), and so the interpretor can execute them in parallel, if possible, but it is not obliged to.

In the definition of Job `G` on line 21, we again see the `args` attribute, though this time we specify more than one argument: one string literal and one variable.

Recall that the attribute `ipdir` tells the interpretor to "top-up" the working directory with more input files while it is running the workflow. In the previous example, this feature was used only in the first Job. In this example, some input files are only specified in later Jobs; this "just-in-time top-up" may allow the interpretor to reduce the amount of data transfered.

In Job `C`, we see an example of the `software_req` attribute. This allows the programmer to stipulate which software packages/libraries are required to run the Job. The interpretor ensures that hosts that cannot provide these packages/libraries will never be assigned this Job. In the current implementations, the information about what packages/libraries are installed on the various nodes are stored in a text file on the local computer. Subsequently, more refined implementations can use some resource-discovery tool (e.g. inGRD [25], Ganglia [27] and NWS [35]) to obtain this information. Note that the program does not explicitly list the Grid nodes for remote execution.

6 Tissue-Specific Gene Expression Example

Human DNA encodes approximately 30,000 to 40,000 genes [22]. A *transcript* is an RNA molecule intermediate between gene and protein in the process known as *The Central Dogma of Molecular Biology*. In organisms such as humans, each gene is divided into sections called exons and introns, of which only the exons are replicated into the transcript.

The human genome has been sequenced, and annotated with known and predicted genes, and can be readily obtained over the internet. Similarly, transcripts isolated from various tissues have also been sequenced and stored in databases available on the internet.

The article in [9] describes an application of GEL in a workflow to analyse how genes are transcribed in different tissues. The workflow extracts the annotated exons from the human genome, which number over 235,000, and compares them, using BLAST [6], against a database of 16,385 transcripts obtained from the Mammalian Gene Collection. That is, the workflow performs over 235,000 searches in a database of 16,385 records. A total of 194 jobs execute over the duration of the workflow.

The current implementation of the workflow executes 120 instances of BLAST in parallel, and requires 6000s to complete on a 64 node cluster of dual 1.4GHz Pentium III machines. Further refinement should yield a shorter makespan, but since the Jobs are submitted through the scheduler, the workflow does not monopolise the cluster, i.e. the scheduler can allow other jobs and/or workflows to run concurrently.

7 Conclusions and Further Work

Bioinformaticians often require compositions of small generic tools to solve specific problems. We believe such solutions can already benefit from current Grid technologies. To this end, we have presented a programming tool which allows such programs to be nat-

urally expressed, and executed on a heterogeneous distributed computer, e.g. a computational Grid. The programming tool provides the following benefits: (1) better load balancing, since scheduling can be performed on a component-by-component/job-by-job basis; (2) cost saving, since fewer expensive software licences and less storage hardware may now be sufficient; and (3) middleware independence, since scripts no longer have middleware-specific references.

Unlike Web Service-based approaches to workflows, we work directly with the program binaries similarly to the GriddLeS project [3]. Thus, it is not necessary to re-engineer existing, well-trusted applications as services (e.g. Web or Grid Services), and it is more convenient when working with components which are still being modified (e.g. custom, in-house applications). Finally, our programs have no middleware-specific references, and so can be run on any middleware with a supporting interpretor, be it Globus, Unicore, PBS, SGE, LSF or the humble OS exec.

So far, we have implemented five interpretor instances, all based on the same builder but with different executors. The executors run DAGs (1) on the same machine, taking advantage of multiple processors if available, (2) on an SGE-based cluster, (3) on an LSF-based cluster, (4) on a PBS-based cluster, and (4) on a Globus Grid (using Globus GRAM/GridFTP commands). A Condor-based Grid executor is being developed. The Globus executor has basic round-robin scheduling, and does not take advantage of any optimisations for file transfers. We can improve performance by adding DAG-level scheduling in the Grid executor, either embedded directly, or via an interface to a scheduler service, such as Condor-G [26], Nimrod [4] or APST [8].

The GEL scripts in Figs. 2 and 5 were created by hand using a text editor. We have since developed a drag-and-drop user interface called *Wildfire* which is targeted at bioinformatics users. In particular, it has pre-defined Job templates for common bioinformatics tools. *Wildfire* simplifies GEL programming by abstracting the concrete syntax from the user.

Recovery for different failures is an important issue in Grid computing. So far, GEL only supports basic recovery, namely when a job fails, the interpretor will try to resubmit the job to different hosts. After exhausting all reasonable alternatives, the whole workflow will fail. Better support for failure recovery will be an interesting, and non-trivial, area of future research.

The programming language can be extended in many ways. Examples include: an expression sublanguage (e.g. addition, multiplication, etc.); a generic lambda-binder for naming values (similar to that in functional languages [19, 31, 23]); a more general Job definition mechanism similar to functions (though we must be careful of recursive definitions); higher-order job parameters; and further checking such as type checking and otherwise.

8 Acknowledgements

The authors would like to thank Tan Chee Meng for his help with the Particle Swarm Optimisation example, Prof. Tapabrata Ray for allowing us to use his code, and various anonymous referees for improvements in this manuscript.

References

[1] European Union DataGrid (EU DataGrid) project.

[2] *R Language Definition*. Available from `http://www.r-project.org`.

[3] D. Abramson and J. Komineni. Interprocess communication in GriddLeS: Grid enabling legacy software. Technical report, School of Computer Science and Software Engineering, Monash University. available from `http://www.csse.monash.edu.au/~davida/griddles/references.htm`.

[4] David Abramson, Rok Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *HPDC*, pages 112–121, 1995.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1988.

[6] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucl. Acids Res.*, 25:3389–3402, 1997.

[7] K. Amin, G. von Laszewski, Mihael Hategan, N. J. Zaluzec, B. Alunkal, and S. Nijsure. GridAnt: A client-controllable workflow system. Technical report, Argonne National Laboratory, 2004. `www.globus.org`.

[8] Francine D. Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, 1996.

[9] Ching-Lian Chua, Francis Tang, Yun-Ping Lim, Liang-Yoong Ho, and Arun Krishnan. Implementing a bioinformatics workflow in a parallel and distributed environment. In *Parallel and Distributed Computing: Applications and Technologies*, volume 3320 of *LNCS*, pages 1–4. Springer, Dec 2005.

[10] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, C. Mendes, Z. Shi, A. YarKhan, and J. Dongarra. New Grid scheduling and rescheduling methods in the GrADS project. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04) – Workshop 10 (NSF NGS Workshop)*, 2004. Available at `http://csdl.computer.org/comp/proceedings/ipdps/2004/2132/11/2132110199%aabs.htm`.

[11] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *4th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer, 1998.

[12] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39, 2003.

[13] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. CUP, 1998.

[14] R. C. Eberhart and J. Kennedy. A new optmizer using particle swarm theory. In *Sixth International Symposium on Micro Machine and Human Science*, pages 39–43. IEEE Service Center, 1995.

[15] S. R. Eddy. Profile hidden markov models. *Bioinformatics*, 14:755–763, 1998.

[16] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. The virtual data grid: A new model and architecture for data-intensive collaboration. In *Conference on Innovative Data Systems Research*, 2003.

[17] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[18] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[19] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.1). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, August 1991.

[20] W. E. Johnstone, D. Gannon, and B. Nitzberg. Grids as production computing environments: The engineering aspects of NASA's Information Power Grid. In *Proceedings 8th International*

Symposium on High Performance Distributed Computing. IEEE Press.

[21] Arun Krishnan, Kuo-Bin Li, and Praveen Issac. Rapid detection of conserved regions in protein sequences using wavelets. *In Silico Biology*, 4(2):133–48, 2004.

[22] E.S. Lander, L.M. Linton, B. Birren, C. Nusbaum, and M.C. Zody. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2001.

[23] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.02: Documentation and user's manual*. INRIA, 7 2001.

[24] Kuo-Bin Li, Praveen Issac, and Arun Krishnan. Predicting allergenic proteins using wavelet transform. *Bioinformatics*, 20:2572–8, Nov 2004.

[25] Ryan Lim, Sebastian Ho, and Arun Krishnan. *in*GRD: A resource discovery framework for the grid. *Submitted for publication*, 2003.

[26] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor: A hunter of idle workstations. In *Proceedings of 8th International Conference on Distributed Computing Systems*, pages 104–111, 1988. Available from `http://www.cs.wisc.edu/condor/publications.html`.

[27] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30(7), 2004.

[28] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982.

[29] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[30] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992.

[31] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[32] Carmen G. Moles, Pedro Mendes, and Julio R. Banga. Parameter estimation in biochemical pathways: A comparison of global optimization methods. *Genome Research*, 13, 2003.

[33] T. Ray, K. Tai, and K.C. Seow. An evolutionary algorithm for constrained optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 771–777. Morgan Kaufmann, 2000.

[34] P. Rice, I. Longden, and A. Bleasby. EMBOSS: The European Molecular Biology Open Software Suite. *Trends in Genetics*, 16:276–277, June 2000.

[35] Richard Wolski. Dynamically forecasting network performance using the Network Weather Service. *Cluster Computing*, 1(1):119–132, 1998.

```
 1: #!/bin/bash
 2: declare -i i
 3: i=1
 4: echo "./prog1 input" | qsub -cwd -N J${i}
 5: i=i+1
 6: holdlist= # null
 7: for ((j=1; j < 11; j++)); do
 8:         str=J${i}
 9:         i=i+1
10:         if [ -z $holdlist ]; then
11:                 holdlist="${str}"
12:         else
13:                 holdlist="${holdlist},${str}"
14:         fi
15:         echo "./prog2 ${j}" | qsub -cwd -N ${str} -hold_jid J1
16: done
17: touch J${i}.ilock
18: lockfile="J${i}.ilock"
19: echo "./prog3; rm -f ${lockfile}" | qsub -cwd -N J${i} -hold_jid ${holdlist}
20: i=i+1
21: while [ -f $lockfile ]; do sleep 10; done     # barrier
22: k=1
23: while [ -z "‘./prog7‘"]; do
24:         k=$k+1
25:         i=i+1 # account for prog7
26:         touch J${i}.ilock
27:         lockfile="J${i}.ilock"
28:         prog4="J${i}"
29:         echo "./prog4; rm -f ${lockfile}" | qsub -cwd -N ${prog4}
30:         while [ -f $lockfile ]; do sleep 10; done  # barrier
31:         i=i+1
32:         holdlist= # null
33:         for f in follower_sol.*; do
34:                 str="J${i}"
35:                 i=i+1
36:                 if [ -z $holdlist ]; then
37:                         holdlist="${str}"
38:                 else
39:                         holdlist="${holdlist},${str}"
40:                 fi
41:                 echo "./prog5 ${f}" | qsub -cwd -N ${str} -hold_jid ${prog4}
42:         done
43:         touch J${i}.ilock
44:         lockfile="J${i}.ilock"
45:         echo "./prog6; rm -f ${lockfile}" | qsub -cwd -N J${i} -hold_jid ${holdlist}
46:         i=i+1
47:         while [ -f $lockfile ]; do sleep 10; done     # barrier
48: done
49: i=i+1
50: touch J${i}.ilock
51: lockfile="J${i}.ilock"
52: echo "./prog8; rm -f J${i}.ilock" | qsub -cwd -N J${i}
53: while [ -f $lockfile ]; do sleep 10; done
54: echo "DONE!"
```

Figure 6: Swarm optimisation implemented using bash and SGE. Note explicit handling of lock files, job names and barriers.